




# Sets and Maps

FRIDAY, APRIL 11

---

Today we will discuss two more powerful containers, which store data in an unordered format: sets and maps.

-  Readings: [Text](#) 5.4, 5.5, Class documentation for [Set](#) and [Map](#)
-  [Lecture quiz on Canvas](#)
-  [Lecture video on Canvas](#)

## Lecture Video

Click to sign in and play video

## Contents

1. Challenge Problem: De-Dupe
2. Set Overview
3. The Stanford Set (Code)
4. Key Set Operations and Operators
5. Sortedness of Set Elements
6. Set Iteration with For-Each Loop
7. Set Application: Removing Duplicates
8. **Supplementary Examples!** Removing and Detecting Duplicates (Variations)
9. Set Speediness
10. Map Overview
11. The Stanford Map (Code)
12. Retrieving Values from Maps
13. Pinging Keys That Aren't in a Map: Two Interesting Behaviors
14. Aside: ISBNs and Book Mappings
15. The Keys in Our Maps Are Distinct
16. Associating First Names with Multiple Last Names

17. Map Iteration (Keys and Values)
18. Sortedness of Map Keys
19. **Supplementary Example!** Map Application: Frequency Tracking
20. **Supplementary Example!** Frequently Occurring Words
21. Map Variable Naming Convention
22. Key Map Operations and Operators
23. Set and Map vs. HashSet and HashMap
24. What's next?
25. Exam Prep

### Challenge Problem: De-Dupe

We started class today with a challenge problem:

Write a function that takes a vector of strings and prints each unique string in that vector exactly once. For example, given the following:

```
{"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"}
```

... we would print "unicorn", "cupcake", and "swamp" exactly once.

The main reason I started with this problem was so we could all appreciate the beauty of the `Set` ADT and its inherent ability to de-dupe a collection of elements. We first saw the following solution, which I explained before implementing in code:

 **Warning! This code has some design issues! See the revisions that follow.**

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

void printUnique(Vector<string> v)
{
    for (int i = 0; i < v.size(); i++)
    {
        // Only print v[i] if we have not already seen this string
        // earlier in the vector.
        bool isUnique = true;

        for (int j = i - 1; j >= 0; j--)
        {
            if (v[i] == v[j])
            {
                isUnique = false;
            }
        }

        // If we got through the loop above without setting isUnique
        // to false, we must not have seen this string yet. So, we
        // print it out.
        if (isUnique)
        {
            cout << v[i] << endl;
        }
    }
}

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    printUnique(v);

    return 0;
}
```

**output:**

```
unicorn
cupcake
swamp
```

We observed two shortcomings of the approach above:

1. Firstly, the vector is passed by value. If we have a ton of data in that vector, we can potentially save a lot of time and space by passing the parameter by reference instead.
2. Secondly, the approach is a bit complex, and the `printUnique()` function might not be super readable. We can make the function a bit more readable through functional decomposition. Specifically, I outsourced the search for `v[i]` to a separate function that checks whether a vector contains a given value within a specified range of indices.

The resulting code is as follows:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

// Warning: This function assumes 'start' and 'end' are valid indices in
// the given vector. Violating this assumption could crash the program.
bool containedInRange(Vector<string>& v, string target, int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        if (v[i] == target)
        {
            return true;
        }
    }

    return false;
}

void printUnique(Vector<string>& v)
{
    for (int i = 0; i < v.size(); i++)
    {
        if (!containedInRange(v, v[i], 0, i - 1))
        {
            cout << v[i] << endl;
        }
    }
}

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    printUnique(v);

    return 0;
}
```

output:

```
unicorn
cupcake
swamp
```

## Set Overview

After that warmup, we continued our discussion of ADTs today, picking up straight away with sets. A set is a collection that does not allow duplicates or preserve the order in which elements are inserted. Let me unpack that statement a bit:

- Just like the sets we're familiar with from math classes, a set data structure does not allow duplicate elements. If we insert the same element into some set multiple times, the set still only contains one copy of that element. Thus, a set only ever contains either zero copies or one copy of some element.
- A set does not maintain the order in which elements are inserted; if we iterate over all the elements in a set, they are not guaranteed to be in the same order in which they were insert. There is no notion of each element having an index (as with vectors) and there is not necessarily a notion of an element's relative positioning within a set.

(*Key take-away!*) Thus, we can think of sets primarily as binary membership devices: an element is either a member of a set, or it isn't.

### The Stanford Set (Code)

To create a set using the Stanford C++ Libraries, we must include the following:

```
#include "set.h"
```

The syntax for creating a set is as follows. As with the other ADTs we've seen so far (vectors, grids, stacks, and queues), the Stanford set is a homogenous container. Whenever we create a set, we specify the element type for that set as a **type parameter**:

```
Set< DATA_TYPE > VARIABLE_NAME ;
```

(*New term!*) I have mentioned that we refer to the datatype in the above declaration as a "**type parameter**" or "**template parameter**," but I don't think I got that term into writing in your notes until today.

(*Important note!*) We must capitalize the 'S' in "Set" when declaring one in code. As with the other ADTs we have seen this quarter, C++ has its own built-in version of a set that uses a lowercase 's'.

Here's the first example we saw of an actual set in class today:

```
#include <iostream>
#include "console.h"
#include "set.h"
using namespace std;

int main()
{
    Set<string> set = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};

    cout << set << endl;
    cout << "Set size: " << set.size() << endl;

    return 0;
}
```

**output:**

```
{"cupcake", "swamp", "unicorn"}
Set size: 3
```

Notice that the set ignores our attempts to add multiple copies of "cupcake" and "unicorn". There is absolutely no indication in the set that we attempted to add those strings multiple times or that we only added "swamp" once.

We can also use `add()` and `remove()` to add and remove items from a set like so:

```
#include <iostream>
#include "console.h"
#include "set.h"
using namespace std;

int main()
{
    Set<string> set = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};

    set.add("spindle");
    set.remove("swamp");

    cout << set << endl;
    cout << "Set size: " << set.size() << endl;

    return 0;
}
```

**output:**

```
{"cupcake", "spindle", "unicorn"}
Set size: 3
```

We also have a `contains()` function for checking whether a set contains a particular value. The function returns true or false (not a count, since sets are binary membership devices that do not allow multiple occurrences of any

value).

```
#include <iostream>
#include "console.h"
#include "set.h"
using namespace std;

int main()
{
    Set<string> set = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};

    set.add("spindle");
    set.add("spindle");
    set.add("spindle");
    set.remove("unicorn");

    if (set.contains("unicorn"))
    {
        cout << "Unicorns, hooray!" << endl;
    }
    else
    {
        cout << "Oh no. :( All the unicorns are gone. :( " << endl;
    }

    return 0;
}
```

output:

```
Oh no. :( All the unicorns are gone. :(
```

## Key Set Operations and Operators

Some of the key operations we can perform on sets include:

Member Function	Description
add(value)	adds a value to a set, ignoring it if the set already contains the value
contains(value)	returns <code>true</code> if the set contains the value, <code>false</code> otherwise
remove(value)	removes the value from the set; does nothing if the value is not in the set
size()	returns the number of elements in the set
isEmpty()	returns <code>true</code> if the set is empty, <code>false</code> otherwise

The Stanford set is amenable to a variety of operators that act as aliases for set functions. For example, `set1 - set2` returns a set difference, `set1 * set2` returns the intersection of two sets, `set1 + set2` returns the union of two sets, and `set += value` adds an element to a set. There are also functions corresponding to many of these operations. For an exhaustive list, see: [Stanford Set class](#).

## Sortedness of Set Elements

You might have noticed in the examples above that the contents of our sets always appeared in alphabetical order when we printed them out. Behind the scenes, the Stanford set is using a data structure that keeps the elements in some kind of sorted order so that we can have really fast insertion and look-up operations.

Check out what happens if we add Zoology, though:

```
#include <iostream>
#include "console.h"
#include "set.h"
using namespace std;

int main()
{
    Set<string> set = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};

    set.add("spindle");
    set.remove("swamp");
    set.add("Zoology");

    cout << set << endl;
    cout << "Set size: " << set.size() << endl;

    return 0;
}
```

**output:**

```
{"Zoology", "cupcake", "spindle", "unicorn"}
Set size: 4
```

Our output is no longer in alphabetic order. It *is*, however, sorted using what some people call **ASCIIbetical order**. Recall from [one of our earlier lectures this quarter](#) that characters in C++ are represented using ASCII values. The ASCII values for uppercase letters come before the ASCII values for lowercase letters. (For example: 'A' = 65, 'Z' = 90, 'a' = 97, and 'z' = 122.) The set uses those ASCII values to sort the strings, and so strings that start with uppercase letters end up coming before those that start with lowercase letters.

### Set Iteration with For-Each Loop

We also saw that we can use a for-each loop to run through the elements of a set, and the elements are processed in the same order in which they appear when printing the set using `cout << set` :

```
#include <iostream>
#include "console.h"
#include "set.h"
using namespace std;

int main()
{
    Set<string> set = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};

    set.add("spindle");
    set.remove("swamp");
    set.add("Zoology");
    set.remove("unicorn");

    for (string s : set)
    {
        cout << "-> " << s << endl;
    }

    return 0;
}
```

**output:**

```
-> Zoology
-> cupcake
-> spindle
```

(*Important note!*) Notice that we can't use a traditional for-loop with an index *i* to loop through the set, though, since there is no index associated with each element.

### Set Application: Removing Duplicates

We saw that we can use a set to very easily solve the problem from above of printing all the unique elements in some vector (with no duplicates). We can just toss them into a set and print the set, letting our ADT do all the heavy lifting for us! Here's the code:

```
#include <iostream>
#include "console.h"
#include "set.h"
#include "vector.h"
using namespace std;

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    Set<string> seen;

    for (string s : v)
    {
        seen.add(s);
    }

    cout << seen << endl;

    return 0;
}
```

**output:**

```
{"cupcake", "swamp", "unicorn"}
```

### Supplementary Examples! Removing and Detecting Duplicates (Variations)

*This section contains a variety of supplementary examples for you to review, play with, and explore. Hooray, supplementary examples! Please be sure to glance through these to verify that you're feeling comfortable with your working knowledge of sets.*

Instead of printing the unique elements in a vector, we could print all the *duplicate* elements like so:

```
#include <iostream>
#include "console.h"
#include "set.h"
#include "vector.h"
using namespace std;

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    Set<string> seen;

    for (string s : v)
    {
        if (seen.contains(s))
        {
            cout << "Oo000h! Dupe! " << s << endl;
        }

        // Note that if we see a duplicate in our vector, the following line doesn't
        // add a second copy to the set since sets don't allow duplicates.
        seen.add(s);
    }

    return 0;
}
```

**output:**

```
Oo000h! Dupe! cupcake
Oo000h! Dupe! unicorn
Oo000h! Dupe! unicorn
```

Can you come up with a way to modify the code above so that when it prints all the duplicate elements in some vector, it only prints each duplicate once, no matter how many times it's duplicated in the array? There are many ways to do that, but here's one possible solution:

```
#include <iostream>
#include "console.h"
#include "set.h"
#include "vector.h"
using namespace std;

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    Set<string> seen;
    Set<string> alreadyPrinted;

    for (string s : v)
    {
        if (seen.contains(s))
        {
            // If we haven't printed this one yet, go ahead and print. Then add
            // to our alreadyPrinted set so it won't get printed again.
            if (!alreadyPrinted.contains(s))
            {
                cout << s << endl;
                alreadyPrinted.add(s);
            }
        }

        // Keep track of all the elements we have seen.
        seen.add(s);
    }

    return 0;
}
```

**output:**

```
cupcake
unicorn
```

Here is a different solution to that problem. Do you see how this one is working?

```
#include <iostream>
#include "console.h"
#include "set.h"
#include "vector.h"
using namespace std;

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    Set<string> seen;
    Set<string> dupes;

    for (string s : v)
    {
        if (seen.contains(s))
        {
            // Keep track of each element that we have seen more than once.
            dupes.add(s);
        }

        // Keep track of all the elements we have seen.
        seen.add(s);
    }

    cout << dupes << endl;

    return 0;
}
```

**output:**

```
{"cupcake", "unicorn"}
```

If we wanted to actually *remove* all duplicates from a vector, we could do so by just throwing all its elements into a set, clearing the vector, and dumping the set elements back into the vector, like so:



```

#include <iostream>
#include "console.h"
#include "set.h"
#include "vector.h"
using namespace std;

int main()
{
    Vector<string> v = {"unicorn", "cupcake", "swamp", "cupcake", "unicorn", "unicorn"};
    Set<string> seen;

    for (string s : v)
    {
        seen.add(s);
    }

    v.clear();

    for (string s : seen)
    {
        v.add(s);
    }

    cout << v << endl;

    return 0;
}

```

**output:**

```
{"cupcake", "swamp", "unicorn"}
```

## Set Speediness

(*Key take-away!*) Set operations are *really* fast. We will talk later this quarter about how exactly we measure the general runtime performance of algorithms, but for now, I want you to be aware that these operations are really fast -- much faster than, for example, repeatedly looping through a vector to see if some element occurs there more than once.

## Map Overview

A map is an associative data structure. It maps *keys* to *values* (or, in other words, it *associates* a given key with a value). Each key in a map is distinct and maps to exactly one value. If we try to map a key to a new value, we do not get a second entry in our map; rather, we overwrite the previous association, and the key is now only associated with the new key. Here are some examples:

- We could map the social security numbers of everyone in class to their names. The social security numbers would be the *keys* in the map, and the *values* would be student names. After constructing such a map, we could feed it a social security number, and it would spit out the name of the person associated with that SSN.
- We could map the ISBNs of books to the titles of those books. ISBNs would form the *keys* in the map, and the *values* would be the book names. After constructing such a map, we could feed it an ISBN, and it would spit out the name of the corresponding book. We could create a similar map of ISBNs to author names.

## The Stanford Map (Code)

To create a map using the Stanford C++ Libraries, we must include the following:

```
#include "map.h"
```

The syntax for creating a map is as follows. As with the other ADTs we've seen so far (vectors, grids, stacks, and queues), the Stanford map is a homogenous container, in that the keys must all be of the same type, and the value must all be of the same type (although the key type may be different from the value type). Whenever we create a map, we specify the key and value element types for that map as type parameters:

```
Map< KEY_DATA_TYPE , VALUE_DATA_TYPE > VARIABLE_NAME ;
```

(*Important note!*) We must capitalize the 'M' in "Map" when declaring one in code. As with the other ADTs we have seen this quarter, C++ has its own built-in version of a set that uses a lowercase 'm'.

Here's the first example we saw of an actual map in class today, which maps the first names of our course staff to their last names:

```
#include <iostream>
#include "console.h"
#include "map.h"
using namespace std;

int main()
{
    Map<string, string> map;

    // Here, the keys are "Sean", "Jonathan", and "Yasmine". The values
    // are "Szumlanski", "Coronado", and "Alonso".
    map["Sean"] = "Szumlanski";
    map["Jonathan"] = "Coronado";
    map["Yasmine"] = "Alonso";

    cout << map << endl;

    return 0;
}
```

**output:**

```
{"Jonathan": "Coronado", "Sean": "Szumlanski", "Yasmine": "Alonso"}
```

The above output uses colons to indicate what keys are mapped to which values. The value associated with "Jonathan" is "Coronado". Another way of saying that is that the key "Jonathan" maps to the value "Coronado".

### Retrieving Values from Maps

We saw that one way to look up the value associated with some key is the following:

```
// gets (and prints) the value associated with "Sean"
cout << map["Sean"] << endl;
```

**output:**

```
Szumlanski
```

Another way to achieve that is through the map's `get()` function:

```
// gets (and prints) the value associated with "Sean"
cout << map.get("Sean") << endl;
```

**output:**

```
Szumlanski
```

### Pinging Keys That Aren't in a Map: Two Interesting Behaviors

If we try to look up the value associated with some key that isn't in a map, we get two interesting behaviors:

(*Key take-away!*) Firstly, the Stanford map will return some default value (0 for integers, the empty string for strings, and so on). (See example code below.)

(*Key take-away!*) Secondly, if we use the `map[key]` syntax for a key that is not present in our map, the Stanford map adds that key to the map and associates it with whatever default value it returns! This does **not** happen with the `get(key)` syntax.

For example:

```

#include <iostream>
#include "console.h"
#include "map.h"
using namespace std;

int main()
{
    Map<string, string> map;

    // Here, the keys are "Sean", "Jonathan", and "Yasmine". The values
    // are "Szumlanski", "Coronado", and "Alonso".
    map["Sean"] = "Szumlanski";
    map["Jonathan"] = "Coronado";
    map["Yasmine"] = "Alonso";

    // Below, the stars (*) are printed just to show that there are no characters --
    // not even spaces -- in the empty string results we're getting.

    // This prints an empty string. "Nathan" is not added to the map.
    cout << "result: *" << map.get("Nathan") << "*" << endl;

    // This also prints an empty string, but "Sonia" does get added to the map.
    cout << "result: *" << map["Sonia"] << "*" << endl;

    // Here, we verify the map has changed. "Sonia" has been added, but not "Nathan".
    cout << map << endl;

    return 0;
}

```

**output:**

```

result: **
result: **
{"Jonathan":"Coronado", "Sean":"Szumlanski", "Sonia":"", "Yasmine":"Alonso"}

```

(*Not mentioned in class.*) If desired, we can avoid the appearance that some non-existent key is associated with an empty string (or similar default value) by gating our lookups behind `containsKey()` calls, like so:

```

#include <iostream>
#include "console.h"
#include "map.h"
using namespace std;

int main()
{
    Map<string, string> map;

    // Here, the keys are "Sean", "Jonathan", and "Yasmine". The values
    // are "Szumlanski", "Coronado", and "Alonso".
    map["Sean"] = "Szumlanski";
    map["Jonathan"] = "Coronado";
    map["Yasmine"] = "Alonso";

    if (map.containsKey("Sonia"))
    {
        // We only execute this line if "Sonia" is present in the map as a key. So,
        // there should be no change to the map this time.
        cout << "result: *" << map["Sonia"] << "*" << endl;
    }

    // The map should have only the two original associative mappings. "Sonia" was not
    // added as a key this time.
    cout << map << endl;

    return 0;
}

```

**output:**

```

{"Jonathan":"Coronado", "Sean":"Szumlanski", "Yasmine":"Alonso"}

```

### Aside: ISBNs and Book Mappings

I mentioned briefly that we could map book titles to author names, but the problem with that is that there are many books in the world that have the same title as one another but that were written by different authors. So, a more likely thing to do would be to map ISBNs to book names and author names. A problem there is that the latest ISBN standard uses 13 digits, which a normal `int` in C++ cannot handle. To get around that, we could actually just store 13-digit ISBNs as strings instead of integers.

I should also mention that it's unlikely that we would maintain multiple maps for our books: one for mapping ISBNs to book titles, another for mapping ISBNs to the authors of those books, and so on. We would more likely create a new datatype that would allow us to bundle together a bunch of information about a book (its title, author(s), publisher, publication date, and so on). If we called that datatype `BookInfo`, for example, we would then map ISBN keys to `BookInfo` values. We will actually talk later this quarter about how to bundle data together like that and create new datatypes.

### The Keys in Our Maps Are Distinct

We also saw that the keys in our maps form a set. So, each key in a map is associated with exactly one value. Consider, for example, the following:

```
#include <iostream>
#include "console.h"
#include "map.h"
using namespace std;

int main()
{
    Map<string, string> map;

    // Here, the keys are "Sean", "Jonathan", and "Yasmine". The values
    // are "Szumlanski", "Coronado", and "Alonso".
    map["Sean"] = "Szumlanski";
    map["Jonathan"] = "Coronado";
    map["Yasmine"] = "Alonso";

    // Here, we associate "Julie" with "Zelenski".
    map["Julie"] = "Zelenski";

    // This overwrites the mapping of "Julie" to "Zelenski". The "Julie" key is
    // instead mapped to "Stanford" now, and we lose the old mapping!
    map["Julie"] = "Stanford";

    cout << map << endl;
    cout << "Map size: " << map.size() << endl;

    return 0;
}
```

**output:**

```
{"Jonathan":"Coronado", "Julie":"Stanford", "Sean":"Szumlanski", "Yasmine":"Alonso"}
Map size: 4
```

### Associating First Names with Multiple Last Names

I mentioned above that each key in a map is associated with a single value. If you want a key to be associated with multiple values, it simply cannot be done. However, the single value that a key maps to could be an *entire data structure* that holds multiple elements. For example, if we have a group of people where multiple people share some first name, we could do something like this:

```
#include <iostream>
#include "console.h"
#include "map.h"
#include "vector.h"
using namespace std;

int main()
{
    Map<string, Vector<string>> map;

    // Super Important:
    // Notice the need for the & to create references in the lines below!

    // We know that initially, "Julie" is not in the map. However, the brackets
    // syntax will automatically associate that key with an empty vector of strings.
    Vector<string>& julieNames = map["Julie"];
    julieNames.add("Zelenski");
    julieNames.add("Stanford");

    Vector<string>& chrisNames = map["Chris"];
    chrisNames.add("Gregg");
    chrisNames.add("Piech");

    Vector<string>& nickNames = map["Nick"];
    nickNames.add("Troccoli");
    nickNames.add("Parlante");

    cout << map << endl;

    // Notice that there are only three values in the map, not size!
    // Each vector is a value, and there are only three vectors here.
    cout << "Map size: " << map.size() << endl;

    return 0;
}
```

output:

```
{"Chris":{"Gregg", "Piech"}, "Julie":{"Zelenski", "Stanford"}, "Nick":{"Troccoli",
"Parlante"}}
Map size: 3
```

Here is an alternative approach that eliminates the intermediary vector variables:

```
#include <iostream>
#include "console.h"
#include "map.h"
#include "vector.h"
using namespace std;

int main()
{
    Map<string, Vector<string>> map;

    // We know map[key] returns a vector. We can just call .add() on it directly!
    map["Julie"].add("Zelenski");
    map["Julie"].add("Stanford");
    map["Chris"].add("Gregg");
    map["Chris"].add("Piech");
    map["Nick"].add("Troccoli");
    map["Nick"].add("Parlante");

    cout << map << endl;

    // Notice that there are only three values in the map, not size!
    // Each vector is a value, and there are only three vectors here.
    cout << "Map size: " << map.size() << endl;

    return 0;
}
```

output:

```
{"Chris":{"Gregg", "Piech"}, "Julie":{"Zelenski", "Stanford"}, "Nick":{"Troccoli",
"Parlante"}}
Map size: 3
```

### Map Iteration (Keys and Values)

As with sets, the keys in a map are not associated with a numeric index, so we can't iterate over a map using a traditional integer-based for-loop. We can, however, use for-each loops to iterate over the keys and values like so:

```
#include <iostream>
#include "console.h"
#include "map.h"
#include "vector.h"
using namespace std;

int main()
{
    Map<string, Vector<string>> map;

    map["Julie"].add("Zelenski");
    map["Julie"].add("Stanford");
    map["Chris"].add("Gregg");
    map["Chris"].add("Piech");
    map["Nick"].add("Troccoli");
    map["Nick"].add("Parlante");

    cout << "Keys:" << endl;
    for (string s : map.keys())
    {
        cout << " - " << s << endl;
    }

    cout << endl << "Values:" << endl;
    for (Vector<string> s : map.values())
    {
        cout << " - " << s << endl;
    }

    return 0;
}
```

**output:**

```
Keys:
- Chris
- Julie
- Nick

Values:
- {"Gregg", "Piech"}
- {"Zelenski", "Stanford"}
- {"Troccoli", "Parlante"}
```

Note that if we iterate over a map directly -- without reference to its `keys()` or `values()` functions -- we get the keys:

```
#include <iostream>
#include "console.h"
#include "map.h"
#include "vector.h"
using namespace std;

int main()
{
    Map<string, Vector<string>> map;

    map["Julie"].add("Zelenski");
    map["Julie"].add("Stanford");
    map["Chris"].add("Gregg");
    map["Chris"].add("Piech");
    map["Nick"].add("Troccoli");
    map["Nick"].add("Parlante");

    for (string s : map)
    {
        cout << s << endl;
    }

    return 0;
}
```

**output:**

```
Chris
Julie
Nick
```

### Sortedness of Map Keys

You might notice that the keys in our map are printed in ASCIIbetical order, just as the elements in our sets. Also

much like our sets, the insertion and lookup operations for the Stanford map are much more efficient than, say, looping through an entire vector to see if some key is present. (More on just how fast those operations are in a few days, when we discuss runtime analysis using Big-O notation.)

### Supplementary Example! Map Application: Frequency Tracking

*This section contains a supplementary example for you to review, play with, and explore. Hooray, supplementary examples! Please be sure to glance through it to verify that you're feeling comfortable with your working knowledge of maps.*

Maps are great for tracking how many times a key occurs in some data set. The following program opens a file (poem.txt) and maps each word in the file to its frequency of occurrence -- a map of strings to integers.

The goal here is not to memorize all the file I/O and string processing functions being used, but rather to gain a general awareness that those things exist in our libraries and -- most importantly -- to see how a map can be used to track word frequencies.

```
#include <iostream>
#include "console.h"
#include "filelib.h"
#include "map.h"
#include "strlib.h"
#include "vector.h"
using namespace std;

bool getFrequencies(string filename, Map<string, int>& map)
{
    ifstream ifs;

    if (!openFile(ifs, filename))
    {
        cout << "Failed to open file in getFrequencies(): " << filename << endl;
        return false;
    }

    Vector<string> lines = readLines(ifs);
    for (string line : lines)
    {
        Vector<string> words = stringSplit(line, " ");
        for (string word : words)
        {
            // Note that map.get(word)++ would not work here, but map[word]++ does.
            map[word]++;

            // We could have done the following, but since map[word] gives us a 0
            // for any key not in the map, the line above accomplishes our goal
            // more succinctly.
            //
            // if (map.contains(word))
            // {
            //     int freq = map.get(word);
            //     map[word] = freq + 1;
            // }
            // else
            // {
            //     map[word] = 1;
            // }
        }
    }

    return true;
}

int main()
{
    Map<string, int> wordToFrequencyMap;
    getFrequencies("poem.txt", wordToFrequencyMap);

    // Print occurrence frequencies.
    for (string s : wordToFrequencyMap.keys())
    {
        cout << s << ": " << wordToFrequencyMap.get(s) << endl;
    }

    return 0;
}
```

**poem.txt**

```
roses are red
butterflies are beautiful
```

output:

```
are: 2
beautiful: 1
butterflies: 1
red: 1
roses: 1
```

Of course, we see the keys are printed in ASCIIbetical order again.

### Supplementary Example! Frequently Occurring Words

*This section contains a supplementary example for you to review, play with, and explore. Hooray, supplementary examples! Please be sure to glance through it to verify that you're feeling comfortable with your working knowledge of maps.*

We could make a minor modification to the code above to, say, print all the words that occur in [Bram Stoker's Dracula](#) more than 100 times:

```
// This constant declaration allows us to avoid a magic number in our code below.
const int FREQUENCY_THRESHOLD = 100;

int main()
{
    Map<string, int> wordToFrequencyMap;
    getFrequencies("dracula.txt", wordToFrequencyMap);

    for (string s : wordToFrequencyMap.keys())
    {
        if (wordToFrequencyMap.get(s) > FREQUENCY_THRESHOLD)
        {
            cout << s << ": " << wordToFrequencyMap.get(s) << endl;
        }
    }

    return 0;
}
```

### Map Variable Naming Convention

(*Not mentioned in class.*) A common naming convention for map variable names in industry is to frame them in terms of what the keys and values represent. For example, if we have a map where the keys are ISBNs and the values are the book titles associated with those ISBNs, we might name such a map `isbnToTitleMap`.

### Key Map Operations and Operators

Some of the key operations we can perform on maps include:

Member Function	Description
<code>clear()</code>	removes all key/value pairs from the map
<code>containsKey(key)</code>	returns <code>true</code> if the map contains a value for the given key, <code>false</code> otherwise
<code>map[key]</code>	returns the value mapped to from the given key; if key is not in the map, <b>adds it</b> with the default value (e.g., <code>0</code> or <code>""</code> )
<code>get(key)</code>	returns the value mapped to from the given key; if key is not in the map, returns the default value for the value type, but does <b>not</b> add it to the map
<code>isEmpty()</code>	returns <code>true</code> if the map contains no key/value pairs (size 0), <code>false</code> otherwise
<code>keys()</code>	returns a vector copy of all keys in the map
<code>map[key] = value</code>	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
<code>put(key, value)</code>	identical to <code>map[key] = value</code>



Member Function	Description
remove(key)	removes any existing mapping for the given key (ignored if the key doesn't exist in the map)
size()	returns the number of key/value pairs in the map
toString()	returns a string representing the map; for example: {"are":2, "beautiful":1}
values()	returns a vector copy of all values in the map

For an exhaustive list, see: [Stanford Map class](#).

### Set and Map vs. HashSet and HashMap

If you've poked around in the [Stanford C++ Library docs](#), you might have noticed that we have a `HashSet` and `HashMap` in addition to the `Set` and `Map` we talked about today. The main difference between those classes (which you do not have to know at this time) is that the `Set` and `Map` keep their elements/keys in sorted order, whereas the `HashSet` and `HashMap` do not. There is a tiny performance hit involved with that, meaning that the `Set` and `Map` operations are slightly slower than those of the `HashSet` and `HashMap`, but all four of these ADTs offer incredibly fast insertion and lookup operations, and the runtime difference between the hash versions and the non-hash version is so tiny that it probably won't be apparent to us at all this quarter in the programs we write.

### What's next?

In our next class, we will discuss formal techniques for runtime analysis (i.e., analyzing how fast our algorithms are) and a widely used notation for expressing runtimes called Big-O notation.

### Exam Prep

1. From scratch and without referring to the notes above, code up the `printUnique()` function we saw at the beginning of class today. The function takes a vector of strings and prints each unique string in that vector exactly once. (Duplicate strings are not printed multiple times.)
2. When we run the program below, we see that the `map[key]` syntax adds an empty vector for each of the keys we ping: "Chris", "Julie", and "Nick". Why do those vectors not contain the last names we attempted to add to them?

```
#include <iostream>
#include "console.h"
#include "map.h"
#include "vector.h"
using namespace std;

int main()
{
    Map<string, Vector<string>> map;

    Vector<string> julieNames = map["Julie"];
    julieNames.add("Zelenski");
    julieNames.add("Stanford");

    Vector<string> chrisNames = map["Chris"];
    chrisNames.add("Gregg");
    chrisNames.add("Piech");

    Vector<string> nickNames = map["Nick"];
    nickNames.add("Troccoli");
    nickNames.add("Parlante");

    cout << map << endl;
    cout << "Map size: " << map.size() << endl;

    return 0;
}
```

output:

```
{"Chris":{}, "Julie":{}, "Nick":{}}
Map size: 3
```

**Highlight for solution to Exercise #2:** We are not using reference variables when retrieving the vectors, so we are actually just getting copies of those vectors in `main()` and operating on those. For the fix, see the working example from the section of lecture notes above titled, "Associating First Names with Multiple Last Names."

3. Consider the following program (from today's lecture notes) and its corresponding output:

```
#include <iostream>
#include "console.h"
#include "map.h"
#include "vector.h"
using namespace std;

int main()
{
    Map<string, Vector<string>> map;

    map["Julie"].add("Zelenski");
    map["Julie"].add("Stanford");
    map["Chris"].add("Gregg");
    map["Chris"].add("Piech");
    map["Nick"].add("Troccoli");
    map["Nick"].add("Parlante");

    cout << "Keys:" << endl;
    for (string s : map.keys())
    {
        cout << " - " << s << endl;
    }

    cout << endl << "Values:" << endl;
    for (Vector<string> s : map.values())
    {
        cout << " - " << s << endl;
    }

    return 0;
}
```

**output:**

```
Keys:
- Chris
- Julie
- Nick

Values:
- {"Gregg", "Piech"}
- {"Zelenski", "Stanford"}
- {"Troccoli", "Parlante"}
```

Modify the program's loops so that it outputs all names using the format `firstName lastName`, like so:

```
Chris Gregg
Chris Piech
Julie Zelenski
Julie Stanford
Nick Troccoli
Nick Parlante
```

4. Consider the `getFrequencies()` function from today's lecture notes. Why would replacing `map[word]++` with `map.get(word)++` cause the code to break?

5. Modify the `getFrequencies()` function so that it inverts the map it produces (mapping frequencies to words rather than mapping words to frequencies) and therefore allows us to print words in ascending order by frequency of occurrence (since the map sorts its keys). In doing so, keep in mind that multiple strings might occur the same number of times in some text. Be sure not to lose any words from the map. (So, you might actually want to map frequencies to *vectors* of strings.)

6. As always, after glancing through today's notes, take at least a 15- to 30-minute break, and then come back to your computer and try to replicate the functionality of those programs without referring back to the notes. (E.g., write a function that removes all duplicate elements from a vector. Write various functions that populate sets and maps, and print the results you get from performing various operations on those data structures. Download the text to Bram Stoker's *Dracula* using the link above in today's notes, and write a program that prints all words that occur in that text more than 100 times.)

7. Be sure to glance at the examples in today's notes labeled "*(Not mentioned in class.)*". There are two small additional detail there that I'd like you to be aware of.

8. Be sure also to glance through the sections of today's notes that are tagged with the **Supplementary Examples!** label for additional examples of how the `Set` and `Map` can be used to solve problems very similar to the ones we explored in class today.
9. As always, the textbook and this week's section handout are chock full of great exercises to reinforce this material.
10. (*Super important!*) It's important to be solving problems and writing code on paper at least once a week so that you feel comfortable writing code on paper when you get into your first exam. You don't want to be caught off guard by that and realize when you take the exam that you've developed too many dependencies on your IDE, and you don't want to be distracted by a feeling that writing code on paper feels foreign to you as you're working to rack up as many points as possible!

---

All course materials © Stanford University 2024. This content is protected and may not be shared, uploaded, or distributed.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-11